# Programación Funcional Avanzada

## Procesamiento de Texto Estructurado

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"



UNIVERSIDAD SIMÓN BOLÍVAR

# Procesar JSON

## La lingua franca de los servicios web

- Formato JSON para intercambio de datos
  - Basado en texto simple.
  - Estructura legible para el humano.
  - Subconjunto propio de JavaScript para serialización.
- Objetos construidos como listas nombre/valor.
  - Nombres – cadenas alfanuméricas.
  - Valores – `null`, cadenas, números, booleanos, arreglos unidimensionales u objetos anidados.
- Siempre hace falta un reconocedor.

Mejor que XML...

UNIVERSIDAD SIMÓN BOLÍVAR

# JSON en Haskell

## Aprovechando el sistema de tipos

- Haskell incluye reconocedores y emisores de JSON – como casi cualquier otro lenguaje.
- `Data.Aeson` – altamente optimizada y fácil de usar.
- `Data.Aeson.Encode.Pretty` – mostrar JSON "bonito"

UNIVERSIDAD SIMÓN BOLÍVAR

# Una encuesta sobre pizza

## . . . de una aplicación web, de un startup, blah

```
data Person = Person {
    firstName  :: !Text,
    lastName   :: !Text,
    age        :: !Int,
    likesPizza :: !Bool
} deriving (Show)
```

- La aplicación manejará los datos internamente en este formato.
- Los desarrolladores del *front-end* necesitan JSON.

UNIVERSIDAD SIMÓN BOLÍVAR

# El API fundamental

`Data.Aeson`

- Codificar JSON hacia un `ByteString`

```
encode :: ToJSON a => a -> ByteString
```

- Decodificar entrada JSON desde un `ByteString`

```
decode :: FromJSON a => ByteString -> Maybe a
```

- Decodificador con descripción de errores

```
eitherDecode :: FromJSON a
             => ByteString -> Either String a
```

- Variantes estrictas en el resultado

```
decode'       :: FromJSON a
              => ByteString -> Maybe a
eitherDecode' :: FromJSON a
              => ByteString -> Either String a
```

UNIVERSIDAD SIMÓN BOLÍVAR

# JSON desde y hacia archivos

## Intercambio de datos local

```
[
   { "firstName"   : "John"
   , "lastName"    : "Doe"
   , "age"         :  24
   , "likesPizza"  :  true
   }
...
]
```

- No tiene por qué estar indentado.
- Podría ser *gigantesco* – lazy `ByteString` FTW!
- Nos encantaría terminar con `[Person]`.

UNIVERSIDAD SIMÓN BOLÍVAR

# JSON desde archivos

## La magia de GHC

```
{-# LANGUAGE OverloadedStrings, DeriveGeneric #-}
import Data.Text
import GHC.Generics

data Person = Person {
    firstName  :: !Text,
    lastName   :: !Text,
    age        :: Int,
    likesPizza :: Bool
} deriving (Show,Generic)

instance FromJSON Person
instance ToJSON Person
```

- ¿ByteString, Text o String? – OverLoadedStrings
- GHC genera instancias FromJSON y ToJSON automáticamente.

UNIVERSIDAD SIMÓN BOLÍVAR

# JSON desde archivos

## Pruebas con valores inmutables

```
> eitherDecode $
    fromString goodjson :: Either String [Person]
Right [Person {firstName = "John", ... },
       Person {firstName = "Rose", ... }]
> eitherDecode $
    fromString badjson :: Either String [Person]
Left "when expecting a Bool, encountered Number instead"
```

- Decodificación requiere `Text` o `ByteString` – por eso `fromString`
- Resultado o error de sintaxis explicativo.

UNIVERSIDAD SIMÓN BOLÍVAR

# JSON desde archivos

## ¿Y desde el archivo?

```
main = do
  d <- (eitherDecode <$> B.readFile "pizza.json")
       :: IO (Either String [Person]
  case d of
    Left err -> putStrLn err
    Right ps -> print ps
```

- En este caso es necesario anotar el tipo deseado –
  en otros programas podría no hacer falta según contexto.
- Aprovechar `Control.Applicative`

UNIVERSIDAD SIMÓN BOLÍVAR

# JSON hacia archivos

## Codificar datos Haskell en JSON

```
import qualified Data.ByteString.Lazy      as B
import qualified Data.ByteString.Lazy.Char8 as B8

person = Person {
  firstName = "Ernesto",
  lastName = "áHernndez-Novich",
  age = 45,
  likesPizza = True
}

> B8.putStrLn $ encode person
{"lastName":"áHernndez-Novich","age":45,
 "firstName":"Ernesto","likesPizza":true}
```

- Tanto `Person` como `[Person]` son codificables.
- Emitir `ByteSring` con la codificación adecuada.

UNIVERSIDAD SIMÓN BOLÍVAR

# ¡Ay, pero que feo!

## A los programas sólo les importa la belleza interna

```
import Data.Aeson.Encode.Pretty

B8.putStrLn $ encodePretty person
{
    "age": 45,
    "firstName": "Ernesto",
    "lastName": "áHernndez-Novich",
    "likesPizza": true
}
```

- Indentado y en orden alfabético.
- `encodePretty'` es configurable – indentación y orden.

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿Y si está en un servicio web?

## Can I haz download?

```
import Network.HTTP.Conduit (simpleHttp)

getJSON :: IO B.ByteString
getJSON = simpleHttp "http://localhost/~emhn/pizza.json"

main = do
  d <- (eitherDecode <$> getJSON)
       :: IO (Either String [Person]
  ...
```

- Conduit modela *streams* perezosos – archivos, HTTP...

UNIVERSIDAD SIMÓN BOLÍVAR

# Más flexibilidad

## Data.Aeson

- Instancias `FromJSON` y `ToJSON` para muchos tipos Haskell.

```
> decode "[1,2,3]" :: Maybe [Int]
Just [1,2,3]
> decode "{\"foo\":1,\"bar\":2}"
  :: Maybe (Map String Int)
Just (fromList [("bar",2),("foo",1)])
```

UNIVERSIDAD SIMÓN BOLÍVAR

# La implantación

## Parsec y librerías especializadas

- Objeto JSON es mapa de claves a valores – `Data.HashMap`

```
type Object = HashMap Text Value
```

- Arreglo (lista) JSON es un vector de valores – `Data.Vector`

```
type Array = Vector Value
```

- Un valor JSON puede ser

```
data Value = Object !Object
           | Array !Array
           | String !Text
           | Number !Scientific
           | Bool !Bool
           | Null
```

- Reconocedor escrito con `AttoParsec` y `Blaze`

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿Y si quiero procesar HTML?

## Porque "razones"

- HTML está definido de manera muy precisa en un DTD – los que escriben HTML suelen ser imprecisos (por no decir piratas).
- Múltiples librerías para manipular HTML y analizar su estructura. – discutiremos `Text.HTML` y `Text.XML`
- Emitir HTML/XML correcto es un problema diferente que atacaremos más adelante.

Descarguemos HTML para procesarlo...

UNIVERSIDAD SIMÓN BOLÍVAR

# Conduit otra vez

## Es realmente práctico

```
import Network.HTTP.Conduit (simpleHttp)
import qualified Data.ByteString.Lazy.Char8 as L

url = "http://www.google.com/search?q=42"

main = L.putStrLn . L.take 500 =<< simpleHttp url
```

- `simpleHttp` – interfaz simple basada en URL
  - Sigue redirecciones automáticamente.
  - Entrega el cuerpo de la respuesta – omite encabezados.
  - Lanza una excepción para los errores.
- `http` – control sobre petición, respuesta, encabezados...
- Obtenemos el HTML en un `ByteString`

UNIVERSIDAD SIMÓN BOLÍVAR

# ¿Qué hacer con el HTML?

## Nos interesan elementos específicos del documento

- Queremos determinar cuántos resultados tiene la consulta – donde dice "about . . . results"
- Esta en un `<div>` identificado como `"resultStats"`
- Construir el DOM (árbol HTML) – `Text.HTML.DOM`
- Recorrer el DOM para encontrar el nodo – `Text.XML.Cursos`

Si han usado XPath, esto será un paseo;
si no, también será un paseo.

UNIVERSIDAD SIMÓN BOLÍVAR

# De ByteString a DOM HTML

## Reconocedor, propiamente dicho

```
import Text.HTML.DOM (parseLBS)
import Text.XML.Cursor (Cursor, attributeIs,
  content, element, fromDocument, child,
  ($//), (&|), (&//), (>=>))

cursorFor :: String -> IO Cursor
cursorFor url = do
  page <- simpleHttp url
  return $ fromDocument $ parseLBS page
```

- parseLBS – de ByteString a DOM HTML (Document)
- fromDocument – de Document a un Cursor que, ¡es un zipper!

UNIVERSIDAD SIMÓN BOLÍVAR

# Búsqueda en el DOM

## Operando sobre el *zipper* `Cursor`

```
findNodes :: Cursor -> [Cursor]
findNodes = element "div"
          >=> attributeIs "id" "resultStatus"
          >=> child
```

- `element` – seleccionar elementos con nombre específico.
- `attributeIs` – filtrar sólo aquellos con el atributo descrito.
- `child` – quedarse con los hijos del nodo en contexto.
- `>=>` – composición Kleisli para Monads (WAT?)

```
(>=>) :: Monad m
      => (a -> m b) -> (b -> m c) -> a -> m c
```

Basta pasar de `Cursor` a texto

UNIVERSIDAD SIMÓN BOLÍVAR

# Extrayendo el contenido

## De múltiples cursores...

```
import qualified Data.Text as T

extractData :: Cursor -> T.Text
extractData = T.concat . content
```

- `content` – tomar los nodos del `Cursor` que tengan texto.
- Como podría haber más de uno, usamos `concat`.

Falta mostrar los resultados
(que podrían ser *varios* en algunos escenarios)

UNIVERSIDAD SIMÓN BOLÍVAR

# Mostrando los resultados

## Combinar y convertir a `String`

```
processData :: [Text] -> IO ()
processData =  putStrLn . T.unpack . T.concat
```

- `concat` – combinar múltiples `Text`.
- `unpack` – Convertir `Text` en `String`

Resta expresar el "flujo" del traductor.

UNIVERSIDAD SIMÓN BOLÍVAR

# El programa principal

## Flujo de datos

```
main = do
  cursor <- cursorFor url
  processData $ cursor $// findNodes $| extractData
```

- (`$//`) – tomar todos los nodos de un `Cursor` que cumplan con el criterio de búsqueda.
- (`$|`) – aplicar la función de extracción a todos los nodos de una lista.
- Infijos, de la misma precedencia, asociando a la izquierda.

UNIVERSIDAD SIMÓN BOLÍVAR

# Quiero saber más...

- RFC-4627 – The application/json Media Type for JSON
- Documentación de `Data.Aeson`
- Documentación de `Text.HTML.Dom`
- Documentación de `Text.XML.Cursor`

UNIVERSIDAD SIMÓN BOLÍVAR